Universidad Carlos III
Sistemas Distribuidos 2024-25
Memoria Ejercicio Evaluable 1
Curso 2024-25

Fecha: **11/10/24**

Grupo: **84**

Grado: **Grado en Ingeniería Informática**

Alumnos:

Jorge Rodrigo Leclercq (100459780)

Jorge Pérez Mora (100472386)

Diseño

El código genera clientes (`app-cliente-*.c`) y un servidor (`servidor-mq.c`) que interactúan entre ellos de manera concurrente mediante colas de mensajes. La funcionalidad que ofrece el servidor es la del manejo de tuplas. Estas tuplas están compuestas por una clave, una cadena de caracteres, un array de números decimales de doble precisión acompañado de un número que indica su longitud y unas coordenadas. El servidor le permite al usuario crear, ver, modificar y eliminar tuplas, entre otros.

Es importante mencionar que tanto el cliente como el servidor se apoyan de varios componentes para poder llevar a cabo estas interacciones. Para empezar, el cliente no interactúa directamente con el servidor ni implementa las funcionalidades descritas con anterioridad, si no que existe un proxy (`claves-proxy-mq.h` y `proxy-mq.c`) que se encarga de esto. El proxy ofrece una API que declara, define y otorga al usuario estos servicios al mismo tiempo que establece una conexión mediante colas de mensaje con el servidor (`send_request(struct request *req)`).

A su vez, el servidor, se apoya de otros elementos (`claves.*`), donde se implementan las funciones y se almacena toda la información de las tuplas en una lista enlazada. Se ha elegido una lista enlazada en vez de un diccionario ya que en muchas ocasiones, es más eficiente tanto para el espacio, al ahorrarse el almacenamiento de la *tabla hash*, como en tiempo, al ahorrarse la computación del *hash*. Para manejar las listas enlazadas, se han implementado las funciones `search_node(int key)` y `add_node(struct Node *new_node)` para buscar e insertar tuplas.

Por último, caben mencionar los archivos `error.*`, que con el uso de los macros facilita el debugging y `mensaje*.h`, que definen las estructuras `request` y `response`. Las estructura request está compuesta por una tupla, el código de operación y una referencia a la cola de mensajería mientras que la estructura response está compuesta por una tupla el estado de la llamada.

El servidor y el cliente se componen de archivos muy similares, teniendo en algunas ocasiones el interfijo `proxy-mq`. Esto se ha hecho así para aportar claridad y modularidad.

Makefile

Este Makefile está diseñado para compilar y ejecutar un servidor y múltiples clientes en un entorno basado en colas de mensajes del sistema. Incorpora reglas para la depuración, cobertura de código, y limpieza de archivos generados. Por último y lo más importante, genera y enlaza con cada uno de los clientes una librería dinámica, `libclaves.so`, a partir de proxy-mq.o y error.o:

Variables y Configuración

`BIN_FILES`: define los ejecutables a generar, incluyendo el servidor y los clientes.

`CC`: compilador utilizado (`gcc`).

`CPPFLAGS`: define el directorio de inclusión para encabezados.

`PEDANTIC_PARANOID_FREAK`: advertencias estrictas y opciones de depuración detalladas.

`REASONABLY_CAREFUL_DUDE`: advertencias razonables con depuración estándar.

`NO_PRAYER_FOR_THE_WICKED`: sin advertencias y optimización máxima.

`LDFLAGS`: indica el directorio de librerías.

`LDLIBS`: especifica librerías a enlazar (`-lpthread` para soporte de hilos).

Reglas de Compilación

`all`: borra archivos previos (`clean`), usa optimización (`NO_PRAYER_FOR_THE_WICKED`), y compila los binarios.

`coverage`: activa generación de cobertura de código con gcov.

`debug`: compila con opciones de depuración (`REASONABLY_CAREFUL_DUDE`).

`pedantic`: aplica advertencias estrictas (PEDANTIC_PARANOID_FREAK).

Compilación de Archivos

Compila archivos objeto (`.o`) con `-fPIC` si forman parte de la librería compartida (`libclaves.so`).

`lib/libclaves.so`: Genera una librería compartida a partir de `proxy-mq.o` y `error.o`.

`servidor`: Compila el servidor enlazando `servidor-mq.o` y `claves.o`.

Los clientes (`app-cliente-*`) se compilan individualmente y enlazan con `libclaves.so`.

Reglas de Limpieza

`clean`: elimina archivos objeto (`.o`), ejecutables y la librería compartida. También limpia colas de mensajes en `/dev/mqueue/` para evitar bloqueos al ejecutar nuevas versiones.

Ejecución

`run-server`: Compila y ejecuta el servidor en segundo plano.

`run-client-%`: Ejecuta un cliente específico.

TL;DR: **para compilar ejecute make** en la terminal, **para ejecutar el servidor ejecute make run-server** en la terminal y **para ejecutar un cliente ejecute make run-client-*** en la terminal, donde * es un número entre el 1 y el 10.

Tests

Los tests se han dividido a través de los archivos app-cliente-*.c, albergando uno cada archivo. Hemos comprobado los casos válidos y los inválidos, como pueden ser los casos en los que la longitud de la cadena de caracteres sea mayor a 255.

`app-cliente-1.c/test_comlplete_1()`: incluye un test para todas las funcionalidades de la API.

`app-cliente-2.c/test_complete_2()`: similar al anterior pero con otros valores.

`app-cliente-3.c/test_set()`: test para la función `set_value()`.

`app-cliente-4.c/test_get()`: test para la función `get_value()` después de la ejecución de `set_value()`.

`app-cliente-5.c/test_modify()`: test para la función `modify_value()` después de la ejecución de `set_value()`.

`app-cliente-6.c/test_delete()`: test para `delete_value()`.

`app-cliente-7.c/test_exist_true()`: se comprueba que un valor existente efectivamente existe.

`app-cliente-8.c/test_exist_false()`: se comprueba que un valor inexistente efectivamente no existe.

`app-cliente-9.c/test_invalid_complete()`: similar a los `test_complete_*()` en cuanto a su estructura, pero alterando el orden, por ejemplo ejecutando la función `get_value()` sobre valores inexistentes.

`app-cliente-10.c/test_invalid_length()`: test en el que se intentan establecer dos tuplas, una con una cadena de caracteres superior a 255 caracteres y un array de números decimales de doble precisión con más de 32 números.